Project Report

Group Number-10

April 12, 2019

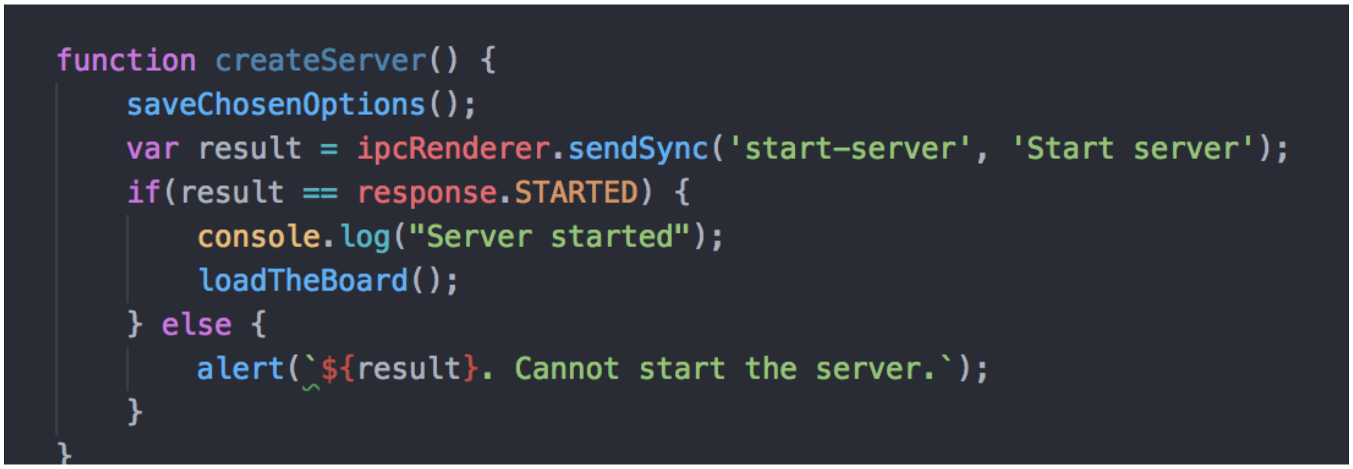
1 Introduction

In our game **Deny and Conquer**, we have used **Electron framework**. The main programming language used is **Javascript**. We used basic **HTML canvas** for the frontend. For the backend, we used dgram library which is an implementation of **UDP datagram sockets**.

2 Architecture

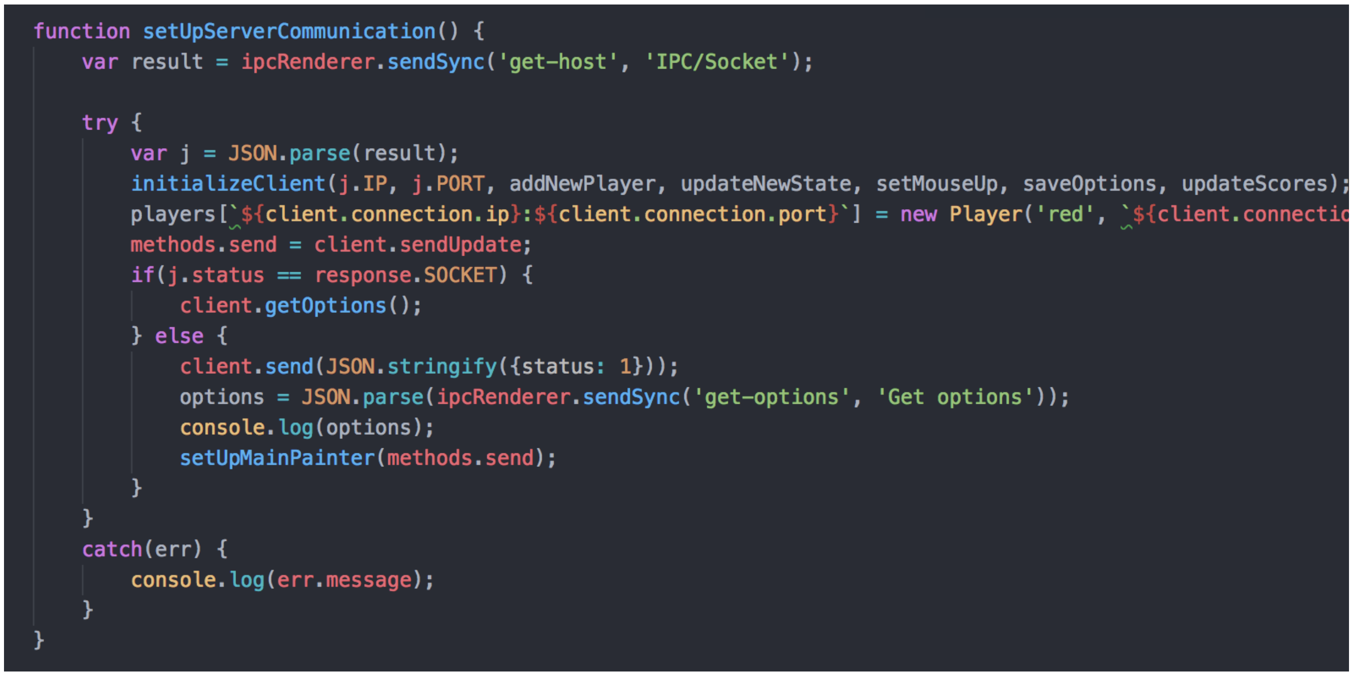
1. We chose a **client-server architecture**. But with the only modification: when a user decides to become a host- it will be a server and a client at the same time. We create 2 UDP sockets in this case - one for the client and another one for the server.

   It can clearly be seen in LoginControl.js and Client.js file:



```
function createServer() {
    saveChosenOptions();
    var result = ipcRenderer.sendSync('start-server', 'Start server');
    if(result == response.STARTED) {
        console.log("Server started");
        loadTheBoard();
    } else {
        alert(`${result}. Cannot start the server.`);
    }
}
```

   It happens when a user chooses to become a host: we save chosen options (for grid size, pen thickness and filing threshold) and tell the main process to create a server. After this Client.js will be loaded:

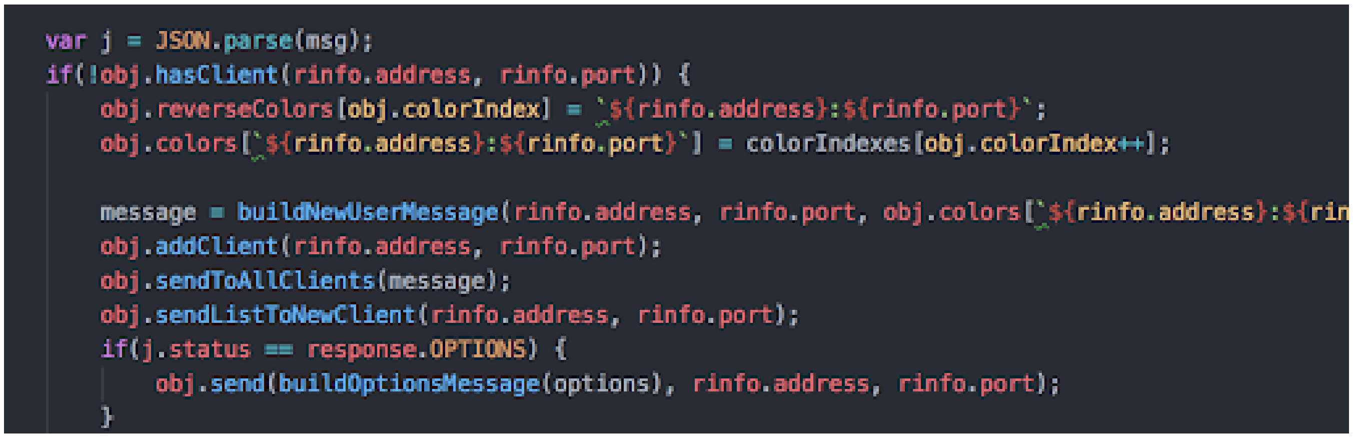


```
function setUpServerCommunication() {
    var result = ipcRenderer.sendSync('get-host', 'IPC/Socket');

    try {
        var j = JSON.parse(result);
        initializeClient(j.IP, j.PORT, addNewPlayer, updateNewState, setMouseUp, saveOptions, updateScores);
        players[`${client.connection.ip}:${client.connection.port}`] = new Player('red', `${client.connectic
        methods.send = client.sendUpdate;
        if(j.status == response.SOCKET) {
            client.getOptions();
        } else {
            client.send(JSON.stringify({status: 1}));
            options = JSON.parse(ipcRenderer.sendSync('get-options', 'Get options'));
            console.log(options);
            setUpMainPainter(methods.send);
        }
    }
    catch(err) {
        console.log(err.message);
    }
}
```

Then we will start Client- create another UDP socket. In the case we are a client - we will get the set options from the server otherwise from main process.

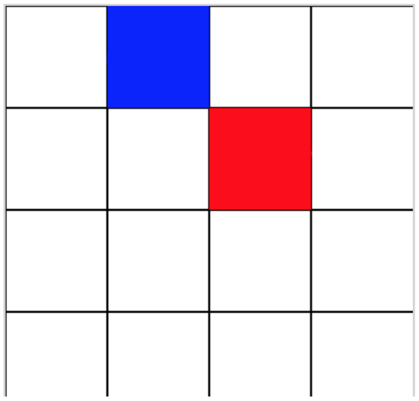
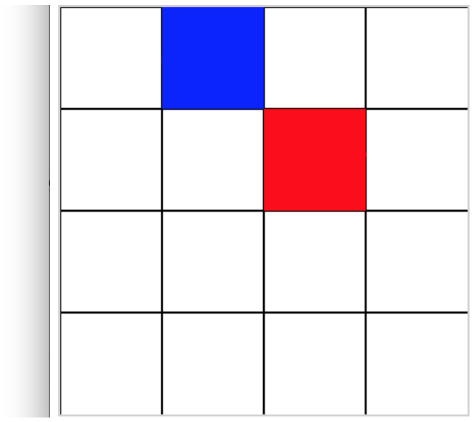
After this the Client and Server communication will begin

2. After a client connects to the server, the server will choose a free color for the user as well as put the user IP and Port in the list of connected users and update all the other users with the information about the new client.

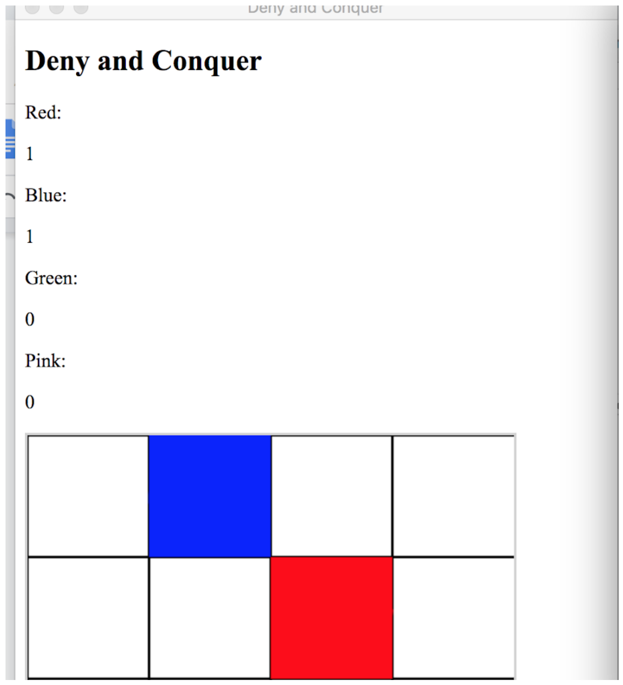


```
var j = JSON.parse(msg);
if(!obj.hasClient(rinfo.address, rinfo.port)) {
    obj.reverseColors[obj.colorIndex] = `${rinfo.address}:${rinfo.port}`;
    obj.colors[`${rinfo.address}:${rinfo.port}`] = colorIndexes[obj.colorIndex++];

    message = buildNewUserMessage(rinfo.address, rinfo.port, obj.colors[`${rinfo.address}:${rin
    obj.addClient(rinfo.address, rinfo.port);
    obj.sendToAllClients(message);
    obj.sendListToNewClient(rinfo.address, rinfo.port);
    if(j.status == response.OPTIONS) {
        obj.send(buildOptionsMessage(options), rinfo.address, rinfo.port);
    }
```

3. Every time a square will be filled-

Information about scores will be updated on the top of the page as shown is the figure below-



Deny and Conquer

Red:

1

Blue:

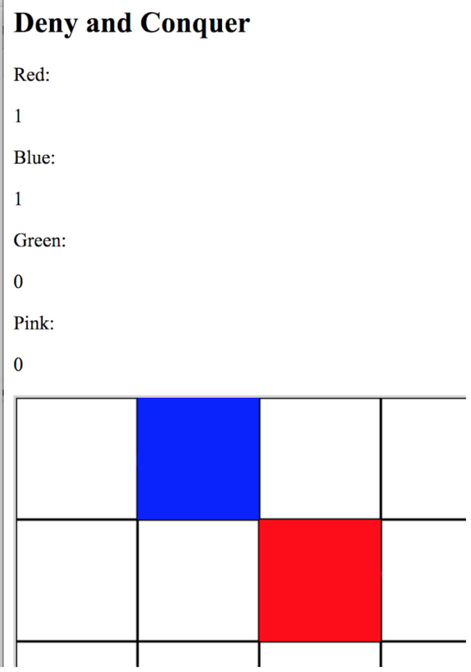
1

Green:

0

Pink:

0



Deny and Conquer

Red:

1

Blue:

1

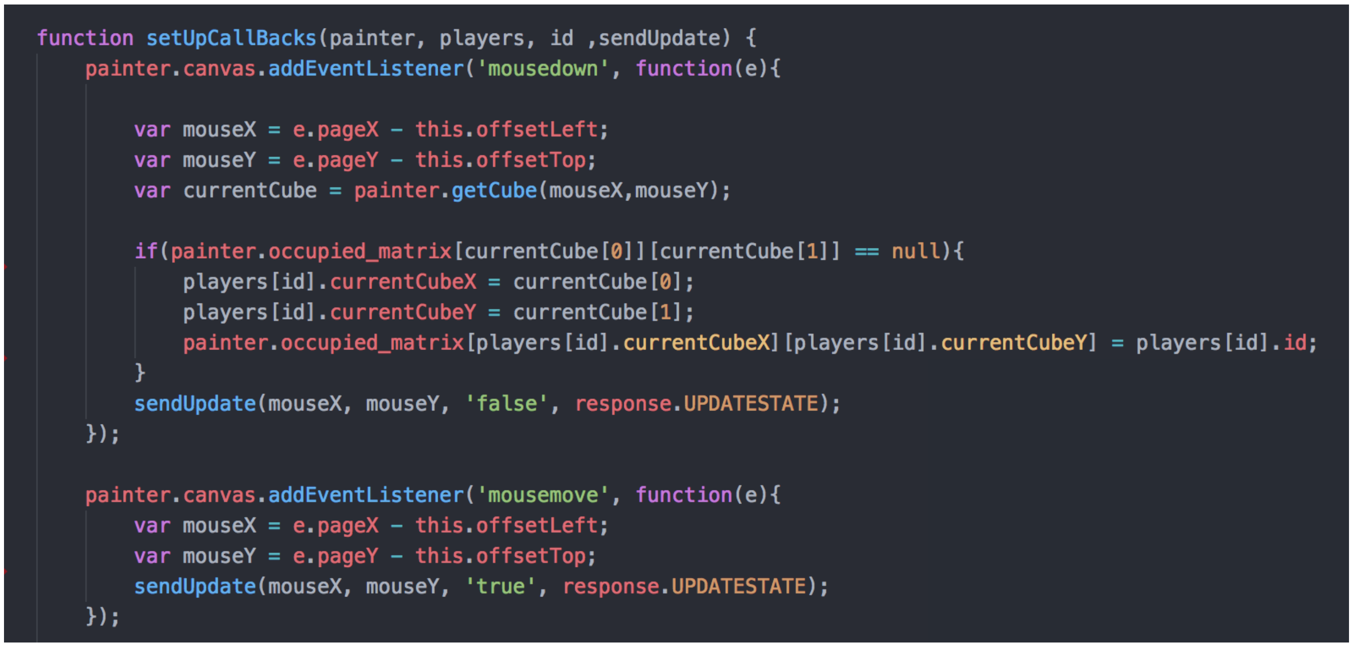
Green:

0

Pink:

0

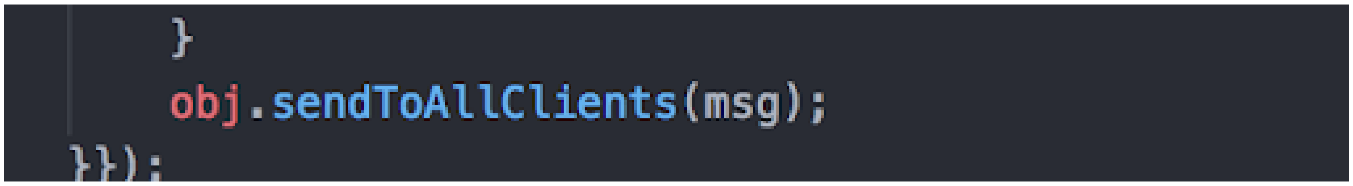
4. After a client makes any move, i.e. any stroke on the screen - information containing the coordinates of the stroke will be sent to the server-



```
function setUpCallBacks(painter, players, id ,sendUpdate) {
    painter.canvas.addEventListener('mousedown', function(e){

        var mouseX = e.pageX - this.offsetLeft;
        var mouseY = e.pageY - this.offsetTop;
        var currentCube = painter.getCube(mouseX,mouseY);

        if(painter.occupied_matrix[currentCube[0]][currentCube[1]] == null){
            players[id].currentCubeX = currentCube[0];
            players[id].currentCubeY = currentCube[1];
            painter.occupied_matrix[players[id].currentCubeX][players[id].currentCubeY] = players[id].id;
        }
        sendUpdate(mouseX, mouseY, 'false', response.UPDATESTATE);
    });

    painter.canvas.addEventListener('mousemove', function(e){
        var mouseX = e.pageX - this.offsetLeft;
        var mouseY = e.pageY - this.offsetTop;
        sendUpdate(mouseX, mouseY, 'true', response.UPDATESTATE);
    });
```

Rendering on the client only happens when this information is being received back by the client so we know that the server has got this message and retransmitted it to all the others. This is how we achieved **concurrency**.

5. **Coordination** is done through maintaining list of clients on the server as well as on every client (for fault tolerance component) and retransmitting the messages to all the connected clients-



```
        }
        obj.sendToAllClients(msg);
}});
```

This function will be executed every time the server receives a new message from any client.

6. When any of the client stops sending messages - server doesn't really care much because we use connectionless UDP. So, the game goes on without any problems. This is how we achieved **Fault Tolerance**. We haven't implemented spinning up a new server feature after the host dies.

3 Conclusion

We were able to achieve **concurrency**, **coordination** and **basic fault tolerance** in our game using the basic socket interactions wrapped in the server and client classes.